1. Subtractive Synthesis Concepts
2. Interactive Time-Varying Digital Filter in LabVIEW
3. Band-Limited Pulse Generator
4. Formant (Vowel) Synthesis
5. Linear Prediction and Cross Synthesis
6. [mini-project] Linear Prediction and Cross Synthesis
7. Karplus-Strong Plucked String Algorithm
8. Karplus-Strong Plucked String Algorithm with Improved Pitch Accuracy

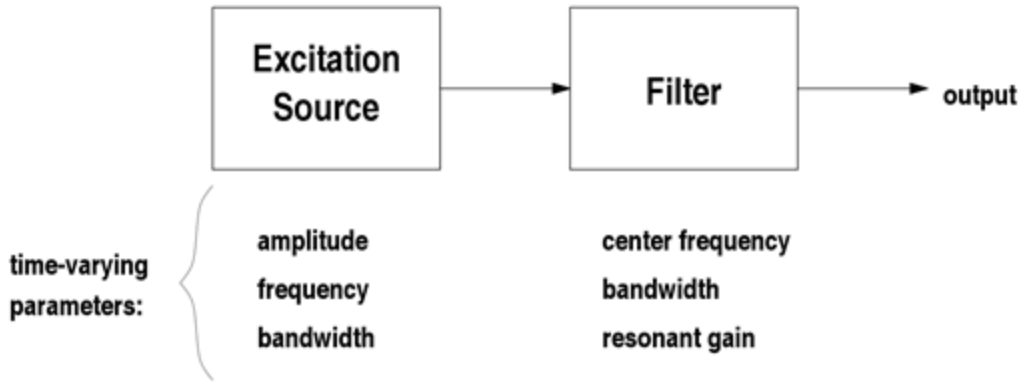
Subtractive Synthesis Concepts

	This module refers to LabVIEW, a software development environment that features a graphical programming language. Please see the LabVIEW QuickStart Guide module for tutorials and documentation that will help you:
	• Apply LabVIEW to Audio Signal Processing
	• Get started with LabVIEW
	• Obtain a fully-functional evaluation edition of LabVIEW

Subtractive Synthesis Model

Subtractive synthesis describes a wide range of synthesis techniques that apply a filter (usually time-varying) to a wideband excitation source such as noise or a pulse train. The filter shapes the wideband spectrum into the desired spectrum. The excitation/filter technique describes the sound-producing mechanism of many types of physical instruments as well as the human voice, making subtractive synthesis an attractive method for **physical modeling** of real instruments.

[link] illustrates the general model of subtractive synthesis. The excitation source parameters may include amplitude, bandwidth, and frequency (for repetitive pulse train input), and each parameter may change independently as a function of time. The filter parameters likewise may vary with time, and include center frequency, bandwidth, and resonant gain.



Model of the subtractive synthesis process

Excitation Sources

Excitation sources for subtractive synthesis must be **wideband** in nature, i.e., they must contain significant spectral energy over a wide frequency range. A **white noise** source is an idealized source that contains constant energy over all frequencies. Practical noise sources do not have infinite bandwidth but can create uniform spectral energy over a suitable range such as the audio spectrum.

Random number generators form the basis of a variety of noise sources on digital computers. For example, the LabVIEW "Signal Processing" palette contains the following noise sources: uniform, Gaussian, random, gamma, Poisson, binomial, and Bernoulli.

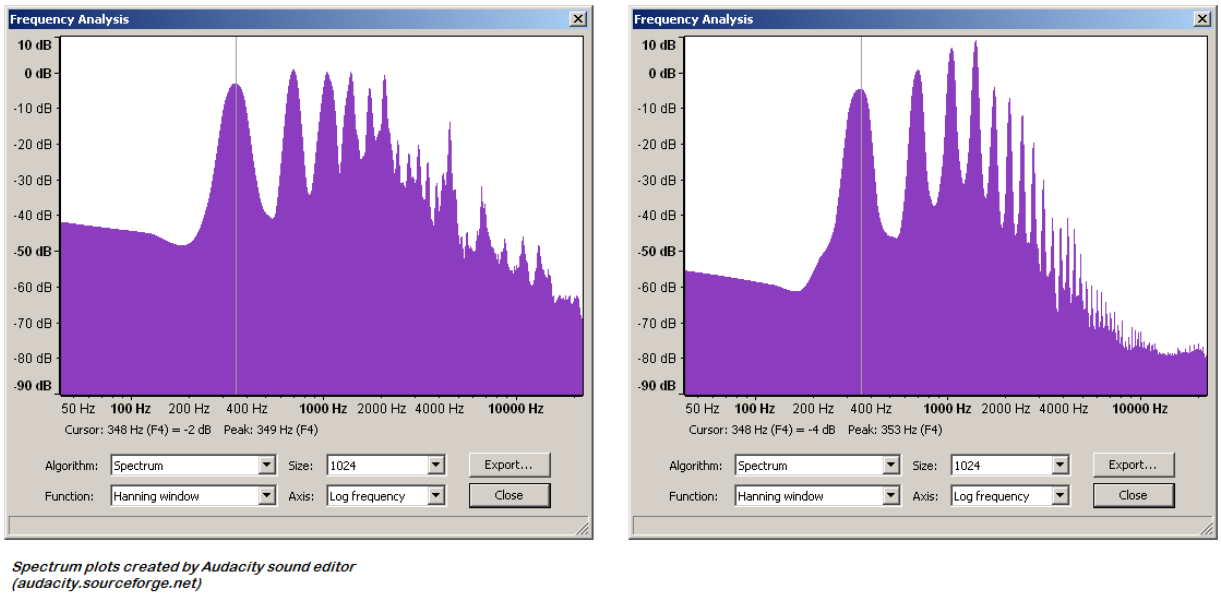
A **pulse train** is a repetitive series of pulses. It provides an excitation source that has a perceptible pitch, so in a sense the excitation spectrum is "pre-shaped" before applying it to a filter. Many types of musical instruments use some sort of pulse train as an excitation, notably wind instruments such as brass (trumpet, trombone, tuba) and woodwinds (clarinet, saxophone, oboe, bassoon). For example, consider the trumpet and its mouthpiece ([link] and [link], respectively). Listen to the "buzzing"

sound of the trumpet mouthpiece alone trumpet_mouthpiece.wav, and compare it to the mouthpiece plus trumpet trumpet.wav. [link] compares the time-domain waveform and frequency spectrum of each sound. Both sounds are the same pitch, so the same harmonics are visible in each. However, the mouthpiece buzz contains more spectral energy at higher frequencies.

Trumpet instrument, a member of the brass family (click picture for larger image)

Trumpet mouthpiece, approximately 3 inches in length (click picture for larger image)

mouthpiece alone **mouthpiece and trumpet**



Spectrum plots created by Audacity sound editor (audacity.sourceforge.net)

Spectra of the trumpet mouthpiece alone and mouthpiece-plus-trumpet signals (click picture for larger image)

Time-Varying Digital Filters

Time-varying digital filters are typically implemented with **block processing**, in which an input signal is subdivided into short blocks (also called **frames**) for filtering. Each frame is processed by a constant-coefficient digital filter. However, the constants change from one frame to the next, thereby creating the effect of a time-varying filter.

The choice of **frame length** involves a trade-off between the rate at which the filter coefficients must change and the amount of time required for the filter's transient response. Higher-order filters require more time to reach steady state, and the frame length should be no shorter than the length of the filter's impulse response.

Digital filters may be broadly classified as either **finite impulse response** (**FIR**) or **infinite impulse response** (**IIR**). The latter type is preferred for

most implementations (especially for real-time filtering) because IIR filters have many fewer coefficients than comparable FIR filters. However, IIR filters have the disadvantage of potential stability problems, especially when finite-precision calculations are used.

The digital filter coefficients usually are calculated independently for each frame. That is, it is generally not possible to calculate only two sets of filter coefficients and then interpolate in between. For example, suppose a digital filter is required to have a cutoff frequency that varies anywhere from 100 Hz to 5,000 Hz. Ideally one would be able to calculate a set of filter coefficients for the 100 Hz filter and another set for the 5,000 Hz filter, and then use linear interpolation to determine the coefficients for any intermediate frequency, i.e., 650 Hz. Unfortunately the interpolation technique does not work. For off-line or batch-type processing, filter coefficients can be computed for each frame. For real-time implementation, the filter coefficients must be pre-computed and stored in a lookup table for fast retrieval.

Download and run the LabVIEW VI filter_coeffs.vi. This VI illustrates why it is generally not possible to interpolate filter coefficients between blocks. Try this: increase the "low cutoff frequency" slider and observe the values of the coefficients. Some coefficients vary monotonically (such as a[1]), but others such as a[2] decrease and then increase again. Still others such as the "b" coefficients remain at a constant level and then begin increasing. You can also try different filter types (highpass, bandpass, bandstop) and filter orders.

Interactive Time-Varying Digital Filter in LabVIEW
A time-varying digital filter can easily be implemented in LabVIEW, and this module demonstrates the complete process necessary to develop a digital filter that operates in real-time and responds to parameter changes from the front panel controls. An audio demonstration of the finished result includes discussion of practical issues such as eliminating click noise in the output signal.

	This module refers to LabVIEW, a software development environment that features a graphical programming language. Please see the LabVIEW QuickStart Guide module for tutorials and documentation that will help you:
	• Apply LabVIEW to Audio Signal Processing
	• Get started with LabVIEW
	• Obtain a fully-functional evaluation edition of LabVIEW

A time-varying digital filter can easily be implemented in LabVIEW. The screencast video of [link] walks through the complete process to develop a digital filter that operates in real-time and responds to parameter changes from the front panel controls. The video includes an audio demonstration of the finished result and discusses practical issues such as eliminating clicks in the output signal.

Download the source code for the finished VI: filter_rt.vi. Refer to TripleDisplay to install the front-panel indicator used to view the signal spectrum.
https://youtu.be/jZBMy28C4Xs (16:44)

https://www.youtube.com/embed/jZBMy28C4Xs?rel=0

[video] Building an interactive real-time digital filter in LabVIEW

Band-Limited Pulse Generator

Subtractive synthesis techniques often require a wideband excitation source such as a pulse train to drive a time-varying digital filter. Traditional rectangular pulses have theoretically infinite bandwidth, and therefore always introduce aliasing noise into the input signal. A band-limited pulse (BLP) source is free of aliasing problems, and is more suitable for subtractive synthesis algorithms. The mathematics of the band-limited pulse is presented, and a LabVIEW VI is developed to implement the BLP source. An audio demonstration is included.

	This module refers to LabVIEW, a software development environment that features a graphical programming language. Please see the LabVIEW QuickStart Guide module for tutorials and documentation that will help you:
	• Apply LabVIEW to Audio Signal Processing
	• Get started with LabVIEW
	• Obtain a fully-functional evaluation edition of LabVIEW

Introduction

Subtractive synthesis techniques apply a filter (usually time-varying) to a wideband excitation source such as noise or a pulse train. The filter shapes the wideband spectrum into the desired spectrum. The excitation/filter technique describes the sound-producing mechanism of many types of physical instruments as well as the human voice, making subtractive synthesis an attractive method for **physical modeling** of real instruments.

A **pulse train**, a repetitive series of pulses, provides an excitation source that has a perceptible pitch, so in a sense the excitation spectrum is "pre-

shaped" before applying it to a filter. Many types of musical instruments use some sort of pulse train as an excitation, notably wind instruments such as brass (e.g., trumpet, trombone, and tuba) and woodwinds (e.g., clarinet, saxophone, oboe, and bassoon). Likewise, the human voice begins as a series of pulses produced by vocal cord vibrations, which can be considered the "excitation signal" to the vocal and nasal tract that acts as a resonant cavity to amplify and filter the "signal."

Traditional rectangular pulse shapes have significant spectral energy contained in harmonics that extend beyond the **folding frequency** (half of the sampling frequency). These harmonics are subject to **aliasing**, and are "folded back" into the **principal alias**, i.e., the spectrum between 0 and . The aliased harmonics are distinctly audible as high-frequency tones that, since undesired, qualify as noise.

The **band-limited pulse**, however, is free of aliasing problems because its maximum harmonic can be chosen to be below the folding frequency. In this module the mathematics of the band-limited pulse are developed, and a band-limited pulse generator is implemented in LabVIEW.

Mathematical Development of the Band-Limited Pulse

By definition, a **band-limited pulse** has zero spectral energy beyond some determined frequency. You can use a truncated Fourier series to create a series of harmonics, or sinusoids, as in [link]:
Equation:

The [link] screencast video shows how to implement [link] in LabVIEW by introducing the "Tones and Noise" built-in subVI that is part of the "Signal Processing" palette. The video includes a demonstration that relates the time-domain pulse shape, spectral behavior, and audible sound of the band-limited pulse.

Download the finished VI from the video: blp_demo.vi. This VI requires installation of the TripleDisplay front-panel indicator.
https://youtu.be/Z8MNNAyn9cQ (11:45)
https://www.youtube.com/embed/Z8MNNAyn9cQ?rel=0

[video] Band-limited pulse generator in LabVIEW
using "Tones and Noise" built-in subVI

The truncated Fourier series approach works fine for off-line or batch-mode signal processing. However, in a real-time application the computational cost of generating individual sinusoids becomes prohibitive, especially when a fairly dense spectrum is required (for example, 50 sinusoids).

A closed-form version of the truncated Fourier series equation is presented in [link] (refer to Moore in "References" section below):
Equation:

where

. The closed-form version of the summation requires only three sinusoidal oscillators yet can produce an arbitrary number of sinusoidal components.

Implementing [link] contains one significant challenge, however. Note the ratio of two sinusoids on the far right of the equation. The denominator sinusoid periodically passes through zero, leading to a divide-by-zero error. However, because the numerator sinusoid operates at a frequency that is N times higher, the numerator sinusoid also approaches zero whenever the lower-frequency denominator sinusoid approaches zero. This "0/0"

condition converges to either N or -N; the sign can be inferred by looking at adjacent samples.

References

- Moore, F.R., "Elements of Computer Music," Prentice-Hall, 1990, ISBN 0-13-252552-6.

Formant (Vowel) Synthesis
Speech and singing contain a mixture of voiced and un-voiced sounds (sibilants like “s”). The spectrum of a voiced sound contains characteristic resonant peaks called formants caused by frequency shaping of the vocal tract. In this module, a formant synthesizer is developed and implemented in LabVIEW. The filter is implemented as a set of parallel two-pole resonators (bandpass filters) that filter a band-limited pulse source.

	This module refers to LabVIEW, a software development environment that features a graphical programming language. Please see the LabVIEW QuickStart Guide module for tutorials and documentation that will help you:
	• Apply LabVIEW to Audio Signal Processing
	• Get started with LabVIEW
	• Obtain a fully-functional evaluation edition of LabVIEW

Introduction

Speech and singing contain a mixture of voiced and un-voiced sounds. Voiced sounds associate with the vowel portions of words, while unvoiced sounds are produced when uttering consonants like "s." The spectrum of a voiced sound contains characteristic resonant peaks called **formants**, and are the result of frequency shaping produced by the **vocal tract** (mouth as well as nasal passage), a complex time-varying resonant cavity.

In this module, a **formant synthesizer** is developed and implemented in LabVIEW. The subtractive synthesis model of a wideband excitation source shaped by a digital filter is applied here. The filter is implemented as a set of parallel two-pole resonators (bandpass filters) that filter a band-limited

pulse. Refer to the modules Subtractive Synthesis Concepts and Band-Limited Pulse Generator for more details.

Formant Synthesis Technique

The [link] screencast video develops the general approach to formant synthesis:
https://youtu.be/QS0iAyXWs5I (3:51)
https://www.youtube.com/embed/QS0iAyXWs5I?rel=0

[video] Formant synthesis technique

The mathematics of the band-limited pulse generator and its LabVIEW implementation are presented in the module Band-Limited Pulse Generator.

The two-pole resonator is an IIR (infinite impulse response) digital filter defined by [link] (see Moore in the "References" section for additional details):
Equation:

$$H\ z \qquad R\ \frac{Rz}{R \quad \theta z \quad R\ z}$$

where $\theta \quad \pi\ f_C\ f_S$, $R \quad e^{\ \pi B\ f_S}$, f_C is the center frequency, B is the bandwidth, and f_S is the sampling frequency, all in units of Hz.

The [link] screencast video shows how to create a subVI that implements the two-pole resonator.
https://youtu.be/GYKxVxv0g_M (7:41)
https://www.youtube.com/embed/GYKxVxv0g_M?rel=0

[video] Implementing the two-pole resonator in LabVIEW

Formants for Selected Vowel Sounds

Peterson and Barney (see "References" section) have compiled a list of formant frequencies for common vowels in American English; refer to [link]:

Phonetic Symbol	Example Word	F_1 (Hz)	F_2 (Hz)	F_3 (Hz)
/ow/	bought	570	840	2410
/oo/	boot	300	870	2240
/u/	foot	440	1020	2240
/a/	hot	730	1090	2440
/uh/	but	520	1190	2390
/er/	bird	490	1350	1690
/ae/	bat	660	1720	2410
/e/	bet	530	1840	2480
/i/	bit	390	1990	2550
/iy/	beet	270	2290	3010

Formant frequencies for common vowels in American English (from Peterson and Barney, 1952)

Formant Synthesizer

The previous sections have laid out all of the pieces you need to create your own formant synthesizer. See if you can set up a LabVIEW VI that pulls the pieces together. The [link] screencast video shows how your finished design might operate. The video also discusses how to choose the relative formant amplitudes and bandwidths, as well as the BLP source parameters.
https://youtu.be/E1c1W-MvYgQ (3:15)
https://www.youtube.com/embed/E1c1W-MvYgQ?rel=0

[video] Formant synthesis LabVIEW VI

References

- Moore, F.R., "Elements of Computer Music," Prentice-Hall, 1990, ISBN 0-13-252552-6.
- Peterson, G.E., and H.L. Barney, "Control Methods Used in a Study of the Vowels," Journal of the Acoustical Society of America, vol. 24, 1952.

Linear Prediction and Cross Synthesis
Linear prediction coding (LPC) models a speech signal as a time-varying filter driven by an excitation signal. The time-varying filter coefficients model the vocal tract spectral envelope. “Cross synthesis” is an interesting special effect in which a musical instrument signal drives the digital filter (or vocal tract model), producing the sound of a “singing instrument.” The theory and implementation of linear prediction are presented in this module.

	This module refers to LabVIEW, a software development environment that features a graphical programming language. Please see the LabVIEW QuickStart Guide module for tutorials and documentation that will help you:
	• Apply LabVIEW to Audio Signal Processing
	• Get started with LabVIEW
	• Obtain a fully-functional evaluation edition of LabVIEW

Introduction

Subtractive synthesis methods are characterized by a wideband excitation source followed by a time-varying filter. **Linear prediction** provides an effective way to estimate the time-varying filter coefficients by analyzing an existing music or speech signal. **Linear predictive coding** (**LPC**) is one of the first applications of linear prediction to the problem of speech compression. In this application, a speech signal is modelled as a time-varying digital filter driven by an **innovations sequence**. The LPC method identifies the filter coefficients by minimizing the prediction error between the filter output and the original signal. Significant compression is possible

because the innovations sequence and filter coefficients require less space than the original signal.

Cross synthesis is a musical adaptation of the speech compression technique. A musical instrument or speech signal serves as the original signal to be analyzed. Once the filter coefficients have been estimated, the innovations sequence is discarded and another signal is used as the filter excitation. For example, a "singing guitar" effect is created by deriving the filter coefficients from a speech signal and driving the filter with the sound of an electric guitar; listen to the audio clips below:

- speech.wav -- Speech signal for digital filter coefficients (audio courtesy of the Open Speech Repository, www.voiptroubleshooter.com/open_speech; the sentences are two of the many phonetically balanced **Harvard Sentences**, an important standard for the speech processing community)
- eguitar.wav -- Electric guitar signal for filter input
- speech_eguitar.wav -- Cross-synthesized result (filter output)

Linear Prediction Theory

The [link] screencast video develops the theory behind linear prediction and describes how the technique is applied to musical signals. Practical issues such as choosing the filter block size and filter order are also discussed.
https://youtu.be/Azud6Xhd03I (10:01)
https://www.youtube.com/embed/Azud6Xhd03I?rel=0

[video] Theory of linear prediction and cross synthesis

Linear Prediction Implementation

The previous section explains how you can use the all-pole filter model to implement cross synthesis. But how are the all-pole filter coefficients actually created?

The LabVIEW MathScript feature includes the function **lpc**, which accepts a signal (1-D vector or array) and the desired filter order and returns an array of filter coefficients. For details, select "Tools | MathScript Window" and type `help lpc`.

[mini-project] Linear Prediction and Cross Synthesis

	This module refers to LabVIEW, a software development environment that features a graphical programming language. Please see the LabVIEW QuickStart Guide module for tutorials and documentation that will help you:
	• Apply LabVIEW to Audio Signal Processing
	• Get started with LabVIEW
	• Obtain a fully-functional evaluation edition of LabVIEW

Objective

Linear prediction is a method used to estimate a time-varying filter, often as a model of a vocal tract. Musical applications of linear prediction substitute various signals as excitation sources for the time-varying filter.

This mini-project will give you chance to develop the basic technique for computing and applying a time-varying filter. Next, you will experiment with different excitation sources and linear prediction model parameters. Finally, you will learn about cross-synthesis.

Prerequisite Modules

If you have not done so already, please study the prerequisite modules Linear Prediction and Cross Synthesis. If you are relatively new to LabVIEW, consider taking the course LabVIEW Techniques for Audio Signal Processing which provides the foundation you need to complete this

mini-project activity, including working with arrays, creating subVIs, playing an array to the soundcard, and saving an array as a .wav sound file.

Deliverables

- All LabVIEW code that you develop (block diagrams and front panels)
- All generated sounds in .wav format
- Any plots or diagrams requested
- Summary write-up of your results

Part 1: Framing and De-Framing

Time-varying filters operate by applying a fixed set of coefficients on short blocks (or "frames") of the signal; the coefficients are varied from one frame to the next. In this part you will develop the basic technique used to "frame" and "de-frame" a signal so that a filter can be applied individually to each frame.

Download and open framing.vi.

The "Reshape Array" node forms the heart of framing and de-framing, since you can reshape the incoming 1-D signal vector into a 2-D array of frames. The auto-indexing feature of the "for loop" structure automatically loops over all of the frames, so it is not necessary to wire a value to the loop termination terminal. You can access the individual frame as a 1-D vector inside the loop structure. Auto-indexing is also used on the loop output to create a new 2-D array, so "Reshape Array" is again used to convert the signal back to a 1-D vector.

Study the entire VI, including the unconnected blocks which you will find useful. Complete the VI so that you can select frame sizes of between 1 and 9. Enable the "Highlight Execution" option, and display your block diagram and front panel simultaneously (press Ctrl-T). Convince yourself that your technique works properly. For example, when you select a frame size of 2, you should observe that the front-panel indicator "frame" displays "0,1",

then "2,3", then "4,5", and so on. You should also observe that the "out" indicator matches the original.

Part 2: Time-Varying Filter Using Linear Prediction

Download the file part2.zip, a .zip archive that contains three VIs: part2.vi, blp.vi (band-limited pulse source), and WavRead.vi (reads a .wav audio file). Complete this VI by creating your own "Framer" and "DeFramer" VIs using the techniques you developed in Part 1.

Create or find a speech-type .wav file to use as a basis for the linear prediction filter. Vary the frame size and filter order parameters as well as the various type of excitation sources. Study the effect of each parameter and discuss your results. Submit one or two representative .wav files.

Part 3: Cross Synthesis

"Cross synthesis" applies the spectral envelope of one signal (e.g., speech) to another signal (e.g., a musical instrument). Find or create a speech signal and use it to generate a time-varying filter. Find or create a music signal and use it as the excitation. The sound files should have the same sampling frequency.

Repeat for a second set of signals. You might also try cross synthesizing two different speech signals or two different music signals.

Show your results, particularly the spectrograms of the two original signals and the spectrogram of the output signal.

Select your favorite result and submit .wav files of the two source signals and the output signal.

Karplus-Strong Plucked String Algorithm
The Karplus-Strong algorithm plucked string algorithm produces remarkably realistic tones with modest computational effort. The algorithm requires a delay line and lowpass filter arranged in a closed loop, which can be implemented as a single digital filter. The filter is driven by a burst of white noise to initiate the sound of the plucked string. Learn about the Karplus-Strong algorithm and how to implement it as a LabVIEW "virtual musical instrument" (VMI) to be played from a MIDI file using "MIDI JamSession."

	This module refers to LabVIEW, a software development environment that features a graphical programming language. Please see the LabVIEW QuickStart Guide module for tutorials and documentation that will help you:
	• Apply LabVIEW to Audio Signal Processing
	• Get started with LabVIEW
	• Obtain a fully-functional evaluation edition of LabVIEW

Introduction

In 1983 Kevin Karplus and Alex Strong published an algorithm to emulate the sound of a plucked string (see "References" section). The Karplus-Strong algorithm produces remarkably realistic tones with modest computational effort.

As an example, consider the sound of a violin's four strings plucked in succession: violin_plucked.wav (compare to the same four strings bowed instead of plucked: violin_bowed.wav). Now compare to the Karplus-Strong version of the same four pitches: ks_plucked.wav.

In this module, learn about the Karplus-Strong plucked string algorithm and how to create a LabVIEW virtual musical instrument (VMI) that you can "play" using a MIDI music file.

Karplus-Strong Algorithm

The [link] screencast video develops the theory of the Karplus-Strong plucked string algorithm, which is based on a closed loop composed of a delay line and a low pass filter. As will be shown, the delay line is initialized with a noise burst, and the continuously circulating noise burst is filtered slightly on each pass through the loop. The output signal is therefore quasi-periodic with a wideband noise-like transient converging to a narrowband signal composed of only a few sinusoidal harmonic components.
https://youtu.be/Z2VWp43yMXQ (4:59)
https://www.youtube.com/embed/Z2VWp43yMXQ?rel=0

[video] Theory of the Karplus-Strong plucked string algorithm

LabVIEW Implementation

The Karplus-Strong algorithm block diagram may be viewed as a **single** digital filter that is excited by a noise pulse. For real-time implementation, the digital filter runs continuously with an input that is normally zero. The filter is "plucked" by applying a burst of white noise that is long enough to completely fill the delay line.

As an exercise, review the block diagram shown in [link] and derive the difference equation that relates the overall output y(n) to the input x(n). Invest some effort in this so that you can develop a better understanding of the algorithm. Watch the video solution in [link] only **after** you have completed your own derivation.
https://youtu.be/3LsMCW4SWK4 (1:47)

https://www.youtube.com/embed/3LsMCW4SWK4?rel=0

[video] Difference equation for Karplus-Strong block diagram

The [link] screencast video shows how to implement the difference equation as a digital filter and how to create the noise pulse. The video includes an audio demonstration of the finished result.
https://youtu.be/iA9mk5r3Ix8 (18:44)
https://www.youtube.com/embed/iA9mk5r3Ix8?rel=0

[video] Building the Karplus-Strong block diagram in LabVIEW

Project Activity: Karplus-Strong VMI

In order to better appreciate the musical qualities of the Karplus-Strong plucked string algorithm, convert the algorithm to a **virtual musical instrument** (**VMI** for short) that can be played by "MIDI Jam Session." If necessary, visit MIDI Jam Session, download the application VI .zip file, and view the screencast video in that module to learn more about the application and how to create your own virtual musical instrument. Your VMI will accept parameters that specify frequency, amplitude, and duration of a single note, and will produce a corresponding array of audio samples using the Karplus-Strong algorithm described in the previous section.

For best results, select a MIDI music file that contains a solo instrument or perhaps a duet. For example, try "Sonata in A Minor for Cello and Bass Continuo" by Antonio Vivaldi. A MIDI version of the sonata is available at the Classical Guitar MIDI Archives, specifically Vivaldi_Sonata_Cello_Bass.mid.

Try experimenting with the critical parameters of your instrument, including sampling frequency and the low-pass filter constant . Regarding sampling frequency: lower sampling frequencies influence the sound in **two** distinct ways -- can you describe each of these two ways?

References

- Moore, F.R., "Elements of Computer Music," Prentice-Hall, 1990, ISBN 0-13-252552-6.
- Karplus, K., and A. Strong, "Digital Synthesis of Plucked String and Drum Timbres," Computer Music Journal 7(2): 43-55, 1983.

Karplus-Strong Plucked String Algorithm with Improved Pitch Accuracy
The basic Karplus-Strong plucked string algorithm must be modified with a continuously adjustable loop delay to produce an arbitrary pitch with high accuracy. An all-pass filter provides a continuously-adjustable fractional delay, and is an ideal device to insert into the closed loop. The delay characteristics of both the lowpass and all-pass filters are explored, and the modified digital filter coefficients are derived. The filter is then implemented as a LabVIEW "virtual musical instrument" (VMI) to be played from a MIDI file using "MIDI JamSession."

	This module refers to LabVIEW, a software development environment that features a graphical programming language. Please see the LabVIEW QuickStart Guide module for tutorials and documentation that will help you:
	• Apply LabVIEW to Audio Signal Processing
	• Get started with LabVIEW
	• Obtain a fully-functional evaluation edition of LabVIEW

Introduction

In the project activity of the prerequisite module Karplus-Strong Plucked String Algorithm, you undoubtedly noticed that the pitch accuracy of the basic Karplus-Strong algorithm needs improvement. For example, listen to the short MIDI test sequence ksdemo.mid rendered to audio with the basic algorithm using a sampling frequency of 20 kHz: ksdemo_20kHz.wav. The individual notes sound reasonable, but when the notes are played simultaneously as a chord, the pitch inaccuracy becomes noticeable. The accuracy gets worse at lower sampling frequencies such as 10 kHz:

ksdemo_10kHz.wav; increasing the sampling frequency improves the accuracy, as at 44.1 kHz: ksdemo_44kHz.wav, however, a discerning ear can still detect some problems.

The pitch of the plucked string tone is determined by the loop time, which must be made continuously variable in order to precisely control the pitch. In the basic algorithm, the length of the delay line determines the loop time, and the delay line can only be varied in integer amounts. The **all-pass filter** will be introduced and studied in this module as a means to introduce an adjustable fractional delay into the loop.

As a preview of the results that you can achieve, consider the same MIDI test sequence rendered to audio using the techniques introduced in this section: ks2demo_10kHz.wav and ks2demo_20kHz.wav.

Lowpass Filter Delay

In the prerequisite module, the loop time was determined to be the product of the delay line length and the sampling interval. The reciprocal of the loop time is the pitch (frequency) of the output signal f_0:
Equation:

$$f_0 = \frac{f_S}{N}$$

where f_S is the sampling frequency in Hz and N is the length of the delay line in samples. Because the delay line length must be an integer number of samples, the pitch cannot be set arbitrarily.

Try the following exercises to explore this concept in more detail.
Exercise:

> **Problem:**
>
> The sampling frequency is 40.00 kHz, and the length of the delay line is 40 samples. What is the pitch of the output signal? If the delay line length is decreased by one sample, what is the new pitch?

Solution:

1000 Hz (40 kHz divided by 40 samples); 1026 Hz

Exercise:

Problem:

The sampling frequency is 10.00 kHz, and the length of the delay line is 10 samples. What is the pitch of the output signal? If the delay line length is decreased by one sample, what is the new pitch?

Solution:

1000 Hz (10 kHz divided by 10 samples); 1111 Hz

For each of the two exercises, the first pitch is exactly the same, i.e., 1000 Hz. However, the change in pitch caused by decreasing the delay line by only one sample is substantial (1026 Hz compared to 1111 Hz). But how perceptible is this difference? The module Musical Intervals and the Equal-Tempered Scale includes a LabVIEW interactive front panel that displays the frequency of each key on a standard 88-key piano. Pitch C6 is 1046 Hz while pitch C♯6 (a half-step higher) is 1109 Hz. These values are similar to the change from 1000 Hz to 1111 Hz caused by altering the delay line length by only one sample, so the change is certainly very audible. The abrupt "jump" in frequency becomes less pronounced at lower pitches where the delay line length is longer.

Flexibility to adjust the overall loop time in a continuous fashion is required to improve pitch accuracy. Moreover, any sources of delay in the loop must be accurately known. So far the delay of the low pass filter has been taken as zero, but in fact the low pass filter introduces a delay of its own.

The [link] screencast video describes first how to calculate the delay of an arbitrary digital filter with transfer function $H(z)$.
https://youtu.be/luD0UxxODso (3:34)
https://www.youtube.com/embed/luD0UxxODso?rel=0

[video] Calculating the delay of a filter given H(z)

In general, therefore, the delay is the negated slope of the filter's phase function, and the delay varies with frequency.

Now, consider the specific low pass filter used in the basic Karplus-Strong algorithm. The filter coefficient "g" will be taken as 0.5, making the filter a true two-point averager:
Equation:

$$H_{\text{LPF}}(z) = \frac{1 + z^{-1}}{2}$$

The [link] screencast video continues the discussion by deriving the delay of the low pass filter of [link]. Several techniques for working with complex numbers in LabVIEW are presented and used to visualize the magnitude and phase response of the filter.
https://youtu.be/sHRvujzuhSQ (5:10)
https://www.youtube.com/embed/sHRvujzuhSQ?rel=0

[video] Calculating the delay of the low pass filter

Because the delay of the low pass filter is always 1/2, the pitch may be expressed more precisely as
Equation:

$$f_0 = \frac{f_S}{N + \frac{1}{2}}$$

While this result more accurately calculates the pitch, it does nothing to address the frequency resolution problem.

All-Pass Filter Delay

Now, consider the **all-pass filter** (**APF**) as a means to introduce a variable and fractional delay into the loop. The all-pass filter has a unit magnitude response over all frequencies, so it does not "color" the signal passing through. However, the all-pass filter has a phase response that is approximately linear for all but the highest frequencies, so it introduces an approximately constant delay. Even better, the slope of the phase response is continuously variable, making it possible to adjust the delay as needed between 0 and 1 samples.

The all-pass filter transfer function is
Equation:

$$H_{\text{APF}}(z) = \frac{C + z^{-1}}{1 + Cz^{-1}}$$

where $|C| < 1$ to ensure stability.

The [link] screencast video continues the discussion by exploring the delay of the all-pass filter of [link] as a function of the parameter C.
https://youtu.be/2cWV1NceJ1U (5:07)
https://www.youtube.com/embed/2cWV1NceJ1U?rel=0

> [video] Calculating the delay of the all-pass filter

Implementing the Pitch-Accurate Algorithm

Including the all-pass filter in the basic Karplus-Strong algorithm allows the loop time to be set to an arbitrary value, making it possible to sound a tone with any desired pitch.

This section guides you through the necessary steps to augment the basic algorithm with an all-pass filter, including the derivation of necessary

equations to calculate the delay line length and the fractional delay. Work through the derivations requested by each of the exercises.

To begin, the pitch of the output signal is the sampling frequency f_S divided by the total loop delay in samples:

Equation:

$$f_0 = \frac{f_S}{N + \frac{1}{2} + \Delta}$$

where Δ is the fractional delay introduced by the all-pass filter.

Exercise:

Problem:

Refer to [link]. Derive a pair of equations that can be used to calculate the length of the delay line N and the value of the fractional delay Δ .

Solution:

$N = \left\lfloor \frac{f_S}{f_0} - \frac{1}{2} \right\rfloor$ (the "floor" operator converts the operand to an integer by selecting the largest integer that is less than the operand);
$\Delta = \frac{f_S}{f_0} - N$

The all-pass filter delay can be approximated by [link] (see Moore):

Equation:

$$\Delta = \frac{1 - C}{1 + C}$$

Exercise:

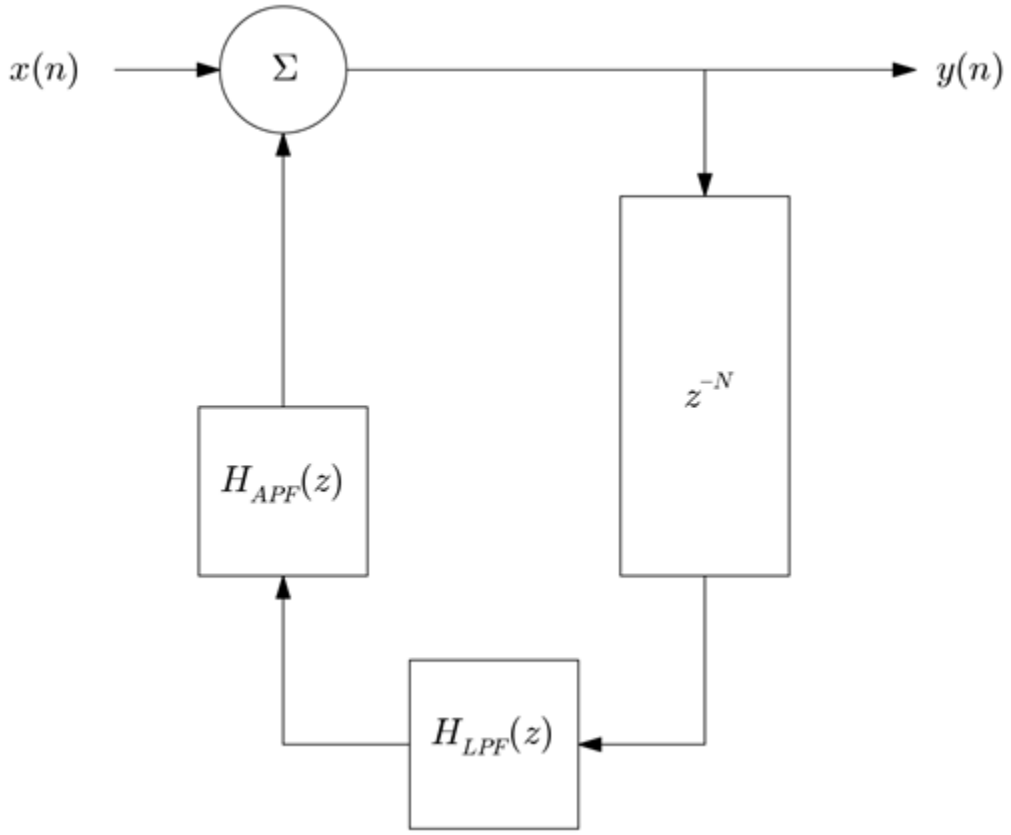
Problem:

Refer to [link]. Solve the equation for the all-pass filter coefficient C in terms of the required fractional delay.

Solution:

$C = \frac{1-\Delta}{1+\Delta}$

The all-pass filter can be inserted at any point in the loop; [link] shows the all-pass filter placed after the low pass filter.



Block diagram of the pitch-accurate Karplus-Strong algorithm

Exercise:

Problem:

To simplify the derivation of the overall filter transfer function that relates y(n) to x(n), consider the three feedback elements (delay line, low pass filter, and all-pass filter) to be a single element with transfer function $G(z)$. Derive the transfer function of the combined element.

Solution:

The elements are in cascade, so the individual transfer functions multiply together: $G(z) = z^{-N} H_{LPF}(z) H_{APF}(z)$

Exercise:

Problem:

Refer to the block diagram of [link]. Considering that all three elements are represented by a single feedback element G(z), derive the overall transfer function H(z) for the digital filter in terms of G(z).

Solution:

$H(z) = \frac{1}{1 - G(z)}$

Recall the low pass filter transfer function ([link]):
Equation:

$$H_{LPF}(z) = g(1 + z^{-1})$$

The all-pass filter transfer function is described by [link].
Exercise:

Problem:

Derive the overall transfer function H(z) in terms of the filter parameters g and C. Write the transfer function in standard form as the ratio of two polynomials in z.

Solution:

$$H(z) = \frac{1+Cz^{-1}}{1+Cz^{-1}-gCz^{-N}-g(1+C)z^{-(N+1)}-gz^{-(N+2)}}$$

Project Activity: Karplus-Strong VMI

As in the prerequisite module, convert the pitch-accurate Karplus-Strong algorithm into a **virtual musical instrument** (**VMI**) that can be played by "MIDI Jam Session." If necessary, visit MIDI JamSession, download the application VI .zip file, and view the screencast video in that module to learn more about the application and how to create your own virtual musical instrument. Your VMI will accept parameters that specify frequency, amplitude, and duration of a single note, and will produce a corresponding array of audio samples using the Karplus-Strong algorithm described in the previous section.

For best results, select a MIDI music file that contains a solo instrument or perhaps a duet. For example, try "Sonata in A Minor for Cello and Bass Continuo" by Antonio Vivaldi. A MIDI version of the sonata is available at the Classical Guitar MIDI Archives, specifically Vivaldi_Sonata_Cello_Bass.mid.

References

- Moore, F.R., "Elements of Computer Music," Prentice-Hall, 1990, ISBN 0-13-252552-6.
- Karplus, K., and A. Strong, "Digital Synthesis of Plucked String and Drum Timbres," Computer Music Journal 7(2): 43-55, 1983.